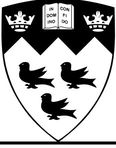

McGill University
School of Computer Science
COMP 621

Optimizing The Optimizer: Improving Data Flow Analysis in Soot

Report No. 2005-07

Michael Batchelder

April 13, 2005

www.cs.mcgill.ca

Contents

1 Introduction and Motivation **3**

2 Background and Related Work **4**
2.1 Sparse Data Flow Graphs 4
2.2 Dominators and Dominance Frontiers 6

3 Specific Problem Statement **6**

4 Solution Strategy and Implementation **6**
4.1 Sparse Flow Analysis 6
4.2 Dominators and Dominance Frontiers 8
4.3 Hand-Optimizations 9

5 Experimental Framework **10**

6 Results **10**
6.1 Experimental Testbed and Environment 10
6.2 Benchmark Programs 10
6.3 Detailed Results and Analysis 11

7 Conclusions and Future Work **17**

8 Acknowledgments **18**

A Appendix **18**
A.1 Source Code and Electronic Versions of this Paper 18
A.2 FastDominatorTreeFrontier.class 18
A.3 FastSparseGraph.class 24

List of Figures

1 Algorithm for constructing a sparse flow graph from Choi, et al. The input is a specific flow analysis domain $D = \langle FG, F, \wedge \rangle$ where FG is the full flow graph and F is the set of flow functions f_i for each node $n_{i_{fg}}$ in FG, and $\wedge$ is the merge operator. K_i denotes the constant result of a function f_i with constant transference. ι denotes the identity result of a function f_i with identity transference. The output of this algorithm is sparse flow graph $SG = \langle N_{sg}, E_{sg} \rangle$ and the mapping $M : E_{fg} \rightarrow N_{sg}$ where N_{sg} denotes the nodes of SG, E_{sg} denotes the edges of SG and E_{fg} denotes the edges of the input FG. 5
2 Cooper, et al's algorithm for computing the dominators of a flow graph FG. Output is D, the mapping from node n to node n's immediate dominator D_n. Note that it is assumed that a topological orderering [2] of nodes is available. 7
3 Cooper, et al's algorithm for computing the dominance frontiers of nodes n in a flow graph FG 7
4 Control flow between sparse flow analysis and it's corresponding sparse flow graph. 9
5 Number of Edges and Nodes in Full flow graphs vs. Number of Edges and Nodes in Sparse flow graphs. 11
6 Cytron vs. Cooper Dominators, Dominator Tree, and Dominance Frontiers Algorithms. . . . 12

7 Jimple to Shimple conversion using Cytron vs. Cooper. 13
8 Forward Flow Analysis, Full vs. Cytron Sparse vs. Cooper Sparse. 14
9 Backward Flow Analysis, Full vs. Cytron Sparse vs. Cooper Sparse 15
10 Improvement on Running Times after Hand Optimizations 16
11 Number of Flow Through and Merge Operations performed, Full vs. Sparse Analysis 17

Abstract

Analysis and optimization tools such as Soot [9] help the optimizing compiler writer explore new analyses, invent new optimizations, and generally better understand the nature of language. Unfortunately, Soot is not always as fast a tool as one would like. The design of the Soot data flow analysis framework focuses on flexibility and generality as well as code readability and maintainability, rather than on speed and optimal memory performance. In this report I explore three areas to try and improve upon these performance limitations. Firstly, I extended Soot's data flow framework to handle generic sparse data flow analysis. Sparse flow analysis iterates to a fixed-point in the same way as full flow analysis yet offers the potential for fewer iterations and less memory requirements. This required the implementation of Choi, et al's [3] algorithm for producing sparse data flow graphs within the context of a flow analysis. Secondly, I implemented the simple and fast dominators and dominance frontiers algorithms from Cooper, et al [4] since this information is required to build sparse flow graphs. These algorithms have worse upper bound running times than more classical approaches by Lengauer and Tarjan [7] and Cytron, et al [5] respectively, but prove faster in almost all real world situations. Finally, I surveyed the Soot flow analysis framework and identified areas for improvement. Relying heavily on key data structure choices and various performance-minded design choices I hand-optimized the code for better time and space performance. In general, I favoured increases in time performance over increases in space performance and I show that these hand optimizations lead to speed gains of a few percentage points. I show through empirical testing that the dominators and dominance frontiers algorithms from Cooper, et al [4] are in fact faster on a wide variety of Java benchmark programs. However, I show that the time to compute this dominance information for building a sparse flow graph pushes the overall time performance of sparse flow analysis above that of full flow analysis within the generic framework of Soot.

1 Introduction and Motivation

Optimizing compilers are important tools for the modern day systems programmer. They can be used to optimize large amounts of code in ways that are far too complex for humans to handle by hand in a timely manner. This can lead to better program performance within the realms of speed, security, memory requirements, program size, and many others. As technology advances and computers get faster one might think that it is a waste of resources to invest time in developing optimizing compilers. However, as technology advances it opens up new and complex realms of problems that computer programmers and users would like to solve. A good example would be video compression - a very common process today for even casual computer users yet nearly unheard of for most ten years ago. Despite the fact that your common computer user may not understand how their computer compresses video, they would still like such a process to finish in a reasonable time frame. There would seem to be a correlation between current technology and current problems: whatever our technology level, we always have more and more complex problems that we would like to solve in as little time as possible. Also, thanks to the same technology that has given us faster computers, there are now more small electronic devices with more functionality and features running more programming languages than ever. These devices need every CPU cycle they can get in order keep up with user expectations. Optimizing compilers can only help.

Analysis and optimization tools such as Soot [9] help the optimizing compiler writer explore new analyses, investigate new ways of looking at programs, improve optimizing code transformations, and generally better understand the nature of language. In short, these tools help in the task of making a good optimizing compiler great; they help to optimize the optimizers.

Unfortunately, Soot is not itself a very fast tool. The language Soot is implemented in, Java, is inherently slower than more low-level alternatives such as C. Programs written in Java are interpreted or compiled just-in-time, suffering some in performance. However, Java has it's own advantages: It is a clear and readable language with built in memory management that is very portable. Therefore it's choice is understandable. Regardless, the faster Soot runs the better.

Building on the idea that compiler optimizations can very often be expressed as a problem within the domain of control flow analysis, I observed that a key component of Soot to improve upon is it's flow analysis framework. To this end, I explore sparse flow analysis in this paper. Sparse flow analysis attempts to reduce the overall work of a flow analysis by creating a new control flow graph abstraction on which to iterate: the

sparse flow graph. This sparse flow graph is a less verbose and more concise set of nodes and edges specific to the particular flow analysis. During further investigation of this technique it was noted that two non-trivial inputs are required: a control flow graph's dominators and dominance frontier sets. I found an interesting paper by Cooper, et al [4] which looks at two algorithms to produce this very information. Their algorithms, while they do not have the best upper bounds discovered thus far, prove to give fairly fast empirical results, in the average case. I consider these algorithms in this paper as well.

After the implementation process, hand optimization of all of my code was done in the hopes of further improving performance.

2 Background and Related Work

2.1 Sparse Data Flow Graphs

Choi, et al introduce the notion of a sparse data flow graph. They describe this graph as follows:

- *It's nodes are flow graph nodes that **generate** information for the data flow problem, or represent the earliest node where new information must be **combined**.*
- *It's edges directly propagate information to nodes that **use** or **combine** the information.*
- *If a node transfers "constant" information, then no information is combined at it's input and it's output is directly propagated.* [3]

The transfer of constant information is a key idea in Choi, et al's algorithm for sparse graph construction. When a node kills all flow information it is considered to have *constant transference*. An example of this would be a flow analysis to find reaching definitions (described in [6]) for a *specific* variable v in a control flow graph. Any node, n, within this graph which defines v invariably kills all previous definitions. Since this is the only information being propagated, the information flowing out of n will *always* be the same: a pointer to itself as the last definition statement; this is constant transference. It can be observed that no node with constant transference ever needs to have information flowed to it and therefore this node, in the more concise sparse graph representation, needs no edges directed to it (i.e. no parents).

Also important is the idea of *identity transference*. A node has identity transference if it does not effect the flow analysis at all. Using the above example problem of a reaching definitions flow analysis, any node which does not have a definition within it is considered to have identity transference. This is because it neither kills nor generates new information. A possible (incorrect) observation at this point is that these nodes can be completely removed from the sparse graph representation. However, if a node n has multiple parents that each, themselves, do not have identity transference (or the parent's parents do not have identity transference, etc), then n must merge flow information and therefore cannot be removed from the sparse graph. Choi, et al describe a concise and elegant algorithm for finding which nodes must be merge points, using the dominators and the dominance frontier sets of the original full flow graph as seen in figure 1. The concept of dominance was first proposed by Prosser [8]. Although he describes dominance of *boxes* in a *diagram*, I paraphrase him here in the vocabulary of control flow graphs:

> Node i dominates node j in a control flow graph if every path leading from the start node to the end node which passes through box j always passes through box i before it reaches j.

Building on this notion, Cytron, et al define a node's dominance frontier as:

> *... the set of all control flow graph nodes, y, such that b dominates a predecessor of y but does not strictly dominate y.* [5]

Here, *strictly dominate* is equivalent to the meaning of dominates from above. In other words, the concept of a dominance frontier of b is all kin y_i of n where y_i has multiple ancestor branches, only one of which contains n. Using this information of dominators and dominance frontiers, the basic idea is that any node

$BuildSparseGraph(FG, F):$

1. $DT =$ Dominator Tree of FG
2. $DF =$ Dominance Frontier of FG
3. $N_{sg} = \{StartNode\} \cup \{\forall n_i \in N_{fg} \mid f_i \neq \iota\}$
4. $MeetNodes = N_{sg} \cup \{DF(m) \mid \forall m \ni N_{sg}\}$
5. $N_{sg} = N_{sg} \cup MeetNodes$
6. $\forall m \in N_{sg}, InSet_m = \top$
7. $Stack = \{\}$, with operators *push*, *pop*, and *peek*
8. $Search(StartNode)$

$Search(x):$

1. if $x \ni MeetNodes$ then $Link(x)$
2. if $x \in N_{sg}$ then $Stack.push(x)$
3. $\forall s \in Successors(x)$
   (a) $M(e_{x \to s}) = Stack.peek()$
   (b) if $s \in MeetNodes$ then $Link(s)$
4. $\forall c \in DT.getChildren(x), Search(c)$
5. if $x \in N_{sg}$ then $Stack.pop()$

$Link(y):$

1. if $y \in N_{sg}$ and $y \neq StartNode$
   (a) if $f_y \neq K_y$
   (a) $z = Stack.peek()$
2. if $f_z = K_z$ then $InSet_y = InSet_y \wedge K_z$
3. else $E_{sg} = E_{sg} \cup e_{z \to y}$

Figure 1: Algorithm for constructing a sparse flow graph from Choi, et al. The input is a specific flow analysis domain $D = \langle FG, F, \wedge \rangle$ where FG is the full flow graph and F is the set of flow functions f_i for each node $n_{i_{fg}}$ in FG, and $\wedge$ is the merge operator. K_i denotes the constant result of a function f_i with constant transference. ι denotes the identity result of a function f_i with identity transference. The output of this algorithm is sparse flow graph $SG = \langle N_{sg}, E_{sg} \rangle$ and the mapping $M : E_{fg} \to N_{sg}$ where N_{sg} denotes the nodes of SG, E_{sg} denotes the edges of SG and E_{fg} denotes the edges of the input FG.

n which is included in the sparse graph (i.e. does not have identity transference) will flow information to a merge point if n has a non-empty dominance frontier set. In fact, all nodes in the dominance frontier of n will be merge points.

These sparse flow graphs (SG) are smaller than their full flow graph (FG) cousins and also do not store analysis flow sets for the set of nodes, N_{fg}, which are only in the FG. For each node $n_{i_{sg}}$ in N_{sg} a mapping is created as the SG is constructed. This mapping connects $n_{i_{sg}}$ to a directed edge in the FG, $e_{n_{i_{sg}} \rightarrow s_i}$ for each of $n_{i_{sg}}$'s successors in FG. If s_i is not included in the SG, the mapping can be used later to look up edge $en_{i_{sg}} \rightarrow s_i$ and the in and out flow sets of s_i can be expressed as the out flowset of $n_{i_{sg}}$.

Because of these features SGs have smaller memory footprints than FGs. They should also reach a fixed-point solution in less iterations since there are, overall, less nodes to flow through. This sparse flow analysis therefore presented an attractive avenue for potential performance improvements in Soot. Unfortunately, this proved not to be the case.

2.2 Dominators and Dominance Frontiers

Observing that Choi, et al's sparse graph construction algorithm requires the full graph's dominators and dominance frontier sets, as defined by [8] and [5] respectively, I explored faster methods for computing this information. Cooper, et al [4] outlined an algorithm (figure 2) for dominators which is simpler and more easily understandable but has a worse upper bound than the well known method of Lengauer and Tarjan [7]. They claim that, despite it's upper bound, it runs faster in most real world situations. I implemented this algorithm in the hopes that it would indeed be a better performer than the Aho, et al [1] algorithm already implemented in Soot. This ultimately proved to be a correct assumption.

Cooper, et al [4] also discuss a simple and fast method for computing dominance frontier sets of a flow graph (figure 3. Similar to the dominator's algorithm, it has a worse upper bound than the original technique by Cytron, et al [5] but makes claims of having a better average running time in empirical tests. I also implemented this algorithm with good results.

3 Specific Problem Statement

In an attempt to improve the running time and memory usage of Soot I implement a sparse graph construction algorithm similar to Choi, et al's [3]. I extend the flow analysis framework of Soot to handle sparse flow analysis using these sparse graphs. I use this implementation to compare sparse flow analysis and full flow analysis to each other in Soot. The initial hope was that there would be improvements in both running time and memory usage.

I implement different dominators and dominance frontiers algorithms, both modeled after those found in Cooper, et al [4], to try and find this information of a flow graph in a more timely fashion. The impetus of this work is to improve upon the time it takes to create a sparse graph, since sparse graph construction requires this data.

I hand-optimize the Soot flow analysis code to investigate whether there is sufficient performance improvements to be gained from human programming effort. How good is the existing Java compiler optimizations? Can more performance be eeked out with intelligent thinking about the problem or do our compilers produce very fast code?

4 Solution Strategy and Implementation

4.1 Sparse Flow Analysis

Soot's framework provides a fairly complete system on which to build compiler tools. The key component of interest is the flow analysis framework which is a fairly simple interface. It can be extended to implement

$Dominators(FG):$

1. $\forall n \in N_{fg}, D_n = NULL$
2. $D_{StartNode} = StartNode$
3. $Changed = true$
4. while $Changed$
   (a) $Changed = false$
   (b) $\forall n \in N_{fg}$
      i. $P = FG.getPredecessors(n)$
      ii. $i =$ first $p \in P$ where $D_p \neq NULL$
      iii. $\forall p \in P$ where $p \neq i$
         A. if $D_p \neq NULL$ then $i = Intersect(p, i)$
      iv. if $D_p \neq i$
         A. $D_p = i$
         B. $Changed = true$

$Intersect(x, y):$

1. while $x \neq y$
   (a) while $TopologicalOrder(x) < TopologicalOrder(y)$ do $x = D_x$
   (b) while $TopologicalOrder(y) < TopologicalOrder(x)$ do $y = D_y$
2. return x

Figure 2: Cooper, et al's algorithm for computing the dominators of a flow graph FG. Output is D, the mapping from node n to node n's immediate dominator D_n. Note that it is assumed that a topological orderering [2] of nodes is available.

$DominanceFrontiers(D, FG):$

1. $\forall n \in N_{fg}$
   (a) $P = FG.getPredecessors(n)$
   (b) if $P \geq 2$
      i. $\forall p \in P$
         A. $runner = p$
         B. while $runner \neq D_n$
            - $DF_runner = DF_runner \cup n$
            - $runner = D_runner$

Figure 3: Cooper, et al's algorithm for computing the dominance frontiers of nodes n in a flow graph FG

specific flow problems. My strategy was to use this already existing framework as a basis for my sparse flow analysis.

Both forward and backwards flow analysis extend the abstract Soot class *FlowAnalysis*. *FlowAnalysis* is itself an extension of the abstract class *AbstractFlowAnalysis*. Between these two classes, the program has all the functionality needed in a flow analysis outlined in base methods and abstract method declarations. In general, the constructor of a flow analysis is passed a directed graph. This graph can be one of many implementations of the *DirectedGraph* interface. The constructor then calls the *doAnalysis* method which handles the main flow analysis algorithm.

Unfortunately, sparse graphs require information about the specific flow analysis they will be used for and flow analyses require access to the graphs they will analyze. This creates a bit of a circular dependency which must be solved in order to properly implement a sparse flow framework. In order to accomplish this I embedded the sparse graph instantiation call within the sparse flow analysis constructor itself. This allows the sparse graph constructor to be passed a reference to the flow analysis (which it will need) and it also allows the flow analysis to retain a reference to the sparse graph. When the sparse graph is first instantiated it effectively computes only steps 1 through 5 in the BuildSparseGraph algorithm of figure 1. At step 6 the sparse graph needs to initialize the in flow sets of it's nodes. However, in Soot the flow sets are handled and initialized in the flow analysis class, within the *doAnalysis* method, not the flow graph. Fortunately, the doAnalysis requires only the nodes in the sparse graph to accomplish this and not the edges, so the flow analysis class is able to initialize the flow sets with just the sparse graph's "node soup" (no edges exist yet). When the analysis is finished initializing the flow sets it then calls the sparse graph's *initialize* method, which completes the construction of the graph by computing steps 7 and 8. When the sparse graph is fully built, the flow analysis finally performs it's iteration to a fixed-point. This control flow is outlined in 4.

It is worth noting that I did not implement the sparse graph mapping of full graph edges $e_{p \to c}$ to sparse graph nodes as in Choi, et al. Instead I map p and c, separately, to the current top of the stack only if they are not in the sparse graph. This is a simpler abstraction which still allows for retrieval of correct flow information and precludes the need for any sort of edge data structure.

Backwards sparse flow analysis is supported by implementing the abstract *FlowAnalysis* method *isForward*() to return false. The sparse flow analysis calls this method in it's constructor and builds a reverse sparse flow graph if the return value is false. The reverse sparse graph is built by passing it a *ReverseDirectedGraph* which is simply a wrapper class around a normal *DirectedGraph* which returns predecessor sets when successor sets are requested and returns tails when heads are requested, and vice-versa. This way there is no need for both forward and backward sparse flow analysis implementations.

Finally, while support for constant transference was built into my sparse flow analysis implementation, there is really little to no notion of it within the soot framework. Because the code being analyzed is in an intermediate representation none of the original variable names are retained. Also, despite the fact that Choi, et al support the usefulness of their sparse flow analysis with single-variable examples, it is unclear where this approach would be very interesting and I feel it is a fairly useless concept.

4.2 Dominators and Dominance Frontiers

The Cooper, et al dominators and dominance frontiers were very straight forward and fairly easy to implement so I will not go into too much detail here. Both algorithms are run from within the *FastDominatorsTreeFinder* class constructor. This class wraps all functionality of dominators, dominator tree, and dominance frontiers into one object. Since actual dominator sets are never required by the sparse graph construction algorithm (only the immediate dominators are needed) I do not build these sets in the constructor. They are built on the fly and cached as they are requested.

As an additional test on this dominators, dominator tree, and dominance frontier suite of algorithms, I implemented a shimple body builder that uses these fast algorithms instead of the default ones in Soot. *FastShimpleBodyBuilder* requests a *FastDominatorTreeFrontier* from the *FastShimpleFactory*, but otherwise the code remains the same as the traditional approach.

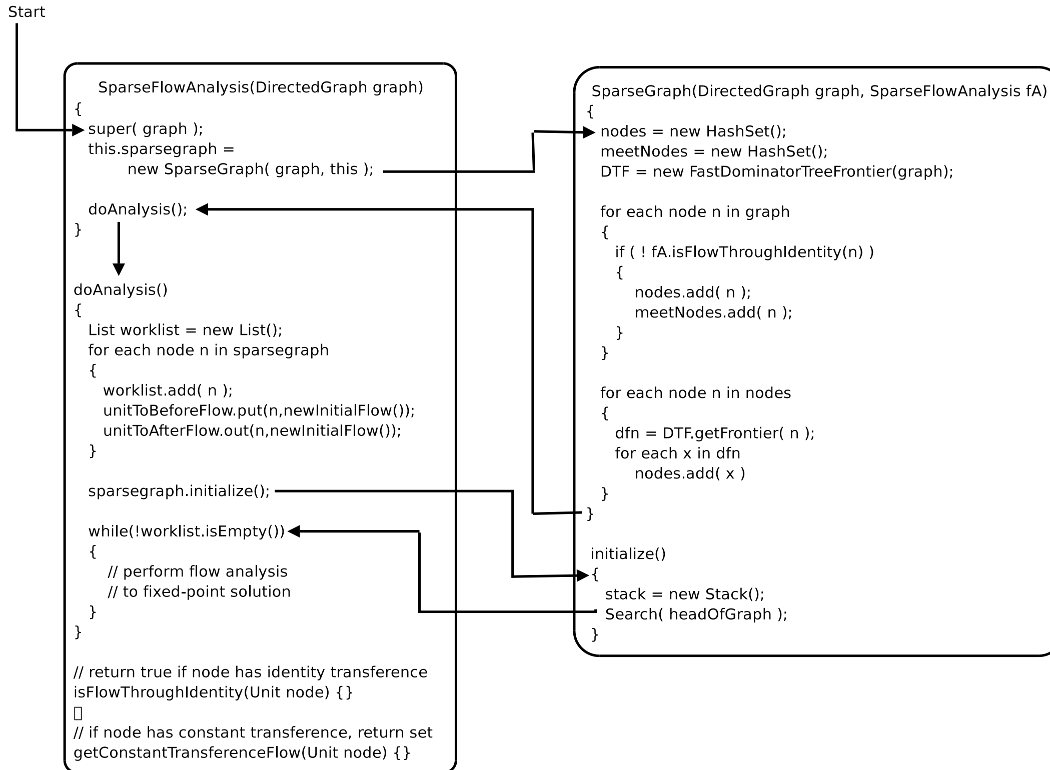

Figure 4: Control flow between sparse flow analysis and it's corresponding sparse flow graph.

4.3 Hand-Optimizations

Convinced that there was yet more performance to be gained I hand-optimized all my code and much of the underlying flow analysis framework in order to get as much out of the system as possible. There is much literature outlining performance tuning in Java, such as Shirazi [10] and I used this as a reference to make good design decisions along the way.

As an example of an optimization that I commonly made, consider a list of flow graph nodes. This set may be a list of heads to the flow graph, a list of predecessors to another node, or a dominance frontier, it is unimportant. Now consider this list being used in the Soot source to iterate over it's elements, to add and remove elements, and to query whether the list contains an element. As is often the case in Soot, the outline I have given here does not require an *ordered* list. Using this last piece of information we can see that a HashSet is a much better choice over an ArrayList as far as algorithm complexity is concerned. Add, get, remove, and contains all have running time $O(1)$ compared to the ArrayList's running times of $O(1)$, $O(n)$, $O(n)$, and $O(n)$, respectively. This is because the ArrayList must maintain an ordering and therefore finding an element in the list means, in the worst case, iterating over the entire list. Certainly, a HashSet may be more memory intensive since it should be initialized to a size greater than the expected number of elements (preferably at least twice expected) to limit the resizing that must occur, but this is an acceptable trade-off in most situations.

Another optimization that is often overlooked is the use of worklists. A worklist can potentially reduce time costs anywhere an algorithm must iterate over a set of elements, processing each element if it meets a certain requirement and potentially creating new elements which meet the requirement. Instead of iterating over all elements over and over until no elements qualify for an entire iteration it is often sufficient to iterate over a worklist. Before the iteration, the worklist is initialized to contain all elements that already meet the requirement. As the iteration is performed, processing that creates new elements which meet the requirements are added to the worklist. This is a well known algorithm optimization.

While I hand-optimized the code for better time *and* space performance, I favoured improvements in running time over decreases in memory usage. These hand optimizations ultimately lead to very little speed gain, measurable in the range of a few percentage points, and the results in the memory usage domain were inconclusive. However, some rare cases resulted in as much as an 8% decrease in running time.

5 Experimental Framework

In order to be sure that all of the code for this project was producing proper results I ran many tests. I implemented a jimple to shimple converter which uses the Cooper, et al dominators and dominance frontier algorithms. All programs in the benchmark suite were converted to shimple using the existing Soot approach and this new approach. The output of these two conversions were compared. All output was identical except for some variable names were flipped. This is still valid output since these variable names are assigned on a first come first serve basis and it is just a matter of different (but legal) orderings of locals. All file sizes between the two conversions were exactly equal.

In addition, two different forward flow analyses and two different backward flow analyses were run on all the benchmark programs. For each flow analysis, two versions were used: one using the traditional full flow analysis and another using the new experimental sparse flow analysis. The flow results were output to xml as source tags and the full and sparse version outputs were compared. Again, other than flip-flopping of flow set elements (which is acceptable, since there was no sense of order in the flow sets I used), the outputs were the same.

The tests outlined in the previous two paragraphs were run again on the final hand-optimized version of the code and verified as well.

6 Results

Unfortunately, as will be seen later in this section, sparse graphs do not end up improving the speed performance of Soot flow analyses. This is a disheartening result but there is still some question as to *why* this is true. The fast dominators and dominance frontiers algorithms of Cooper, et al, however, are an improvement - and a pretty good one. Hand optimized code improvement is minimal.

6.1 Experimental Testbed and Environment

The testbed for this project was an Intel P4 2.66GHz with 512MB Ram running Gentoo Linux. All tests were run on Sun's JVM, version 1.4.2 using a maximum heap size of 400MB. All benchmarks were analyzed 100 times in a loop within the same instantiation of Soot and an average over these 100 trials was taken. Timing was measured using the System.currentTimeMillis() Java call which unfortunately has a fairly large granularity but nevertheless seemed to give good numbers. Unless otherwise noted, all times given are in milliseconds.

6.2 Benchmark Programs

The benchmarks used in this report are a collection of programs created by students of McGill University's COMP-621 Optimizing Compilers class during the Winter 2005 semester. Each benchmark was written to stress-test certain types of operations in Java such as floating point arithmetic, excessive multithreading, and object creation. However, these benchmarks were not used to test Soot's optimization potential. They were only used as a heterogenous collection of control flow graphs for testing the performance of Soot's flow analysis frameworks. Therefore, I will not go into detail on the individual programs in the benchmark suite. The suite of benchmarks can be downloaded from `http://www.cs.mcgill.ca/~mbatch/621/index.html` as a *.tar.gz* file. Each benchmark contains a readme file which explains it's purpose in full detail.

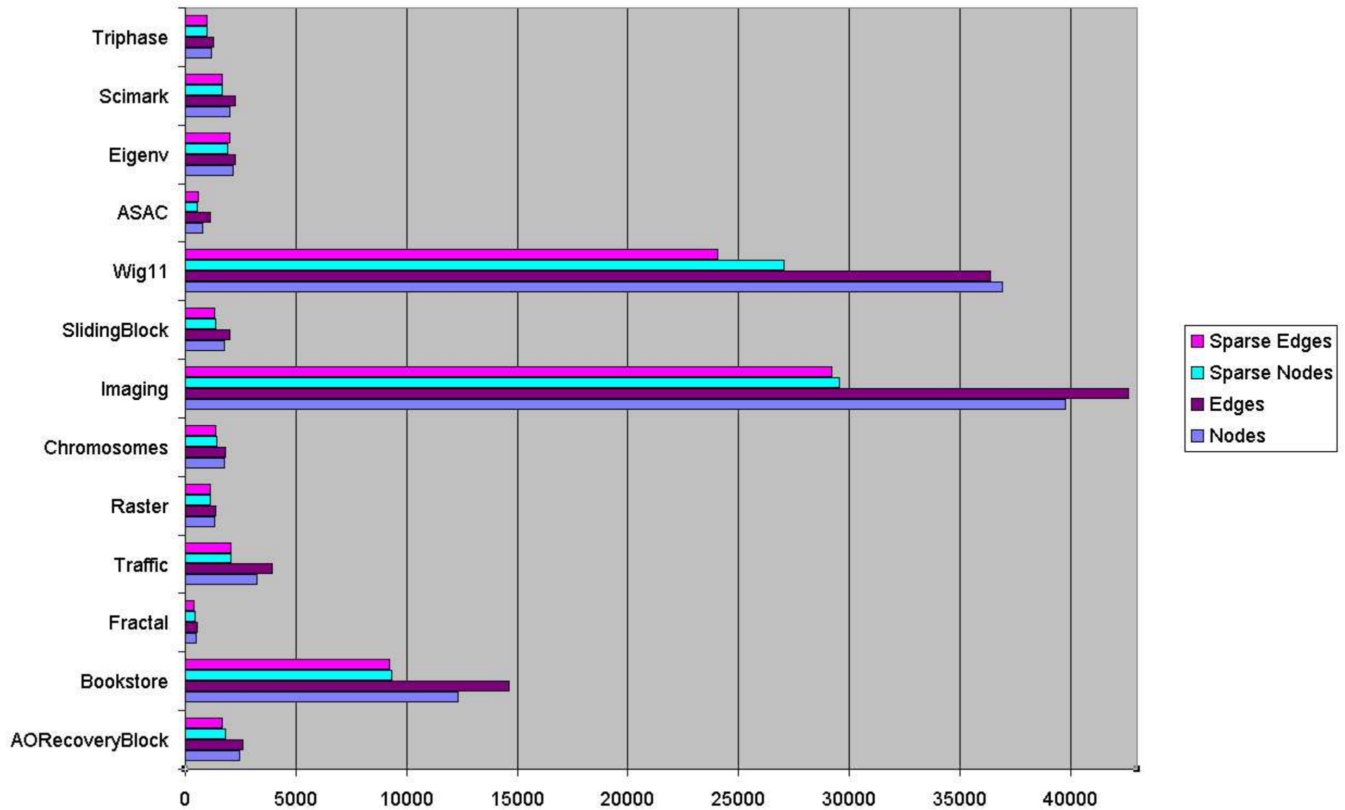

Figure 5: Number of Edges and Nodes in Full flow graphs vs. Number of Edges and Nodes in Sparse flow graphs.

6.3 Detailed Results and Analysis

The first results that were looked at were the sizes of the generated sparse graphs. Were they really smaller than the full graphs? As can be seen in figure 5, the sparse graphs are indeed smaller but they are not *significantly* smaller. This is unfortunate since the cost of producing the dominators and dominance frontier is expensive yet we are not gaining much for it. However, this is somewhat to be expect since we are not allowing for constant transference in our analyses, a feature of sparse flow analysis which presumably helps reduce the sparse graph considerably.

A test of the running times of the Cytron, et al and Ferrante, et al traditional approaches to dominance was compared to that of Cooper, et al's approach. The Cooper approach is indeed faster in all cases and these are great results; they are shown in figure 6.

In order to fully explore the Cytron vs. Cooper results I tested jimple to shimple conversion using the two different approaches. This functionality in Soot is separate from the flow analysis framework and is there a good "second test". Indeed, the results are still good. However, we can see that the impact of the Cooper algorithms is much less since there is a lot of other processing that must be done to convert jimple to shimple. Calculating dominators, dominator tree, and dominance frontiers is only a small subset of the work that has to be done. Also, as was noted early in the implementation section, the dominator sets are not explicitly built in the constructor of the $FastDominatorTreeFrontier$. These sets are computed only when the information is requested, which explains why the Cooper algorithm appears to perform so much better in figure 6: The Cytron algorithm is computing dominator sets where the Cooper algorithm is not.

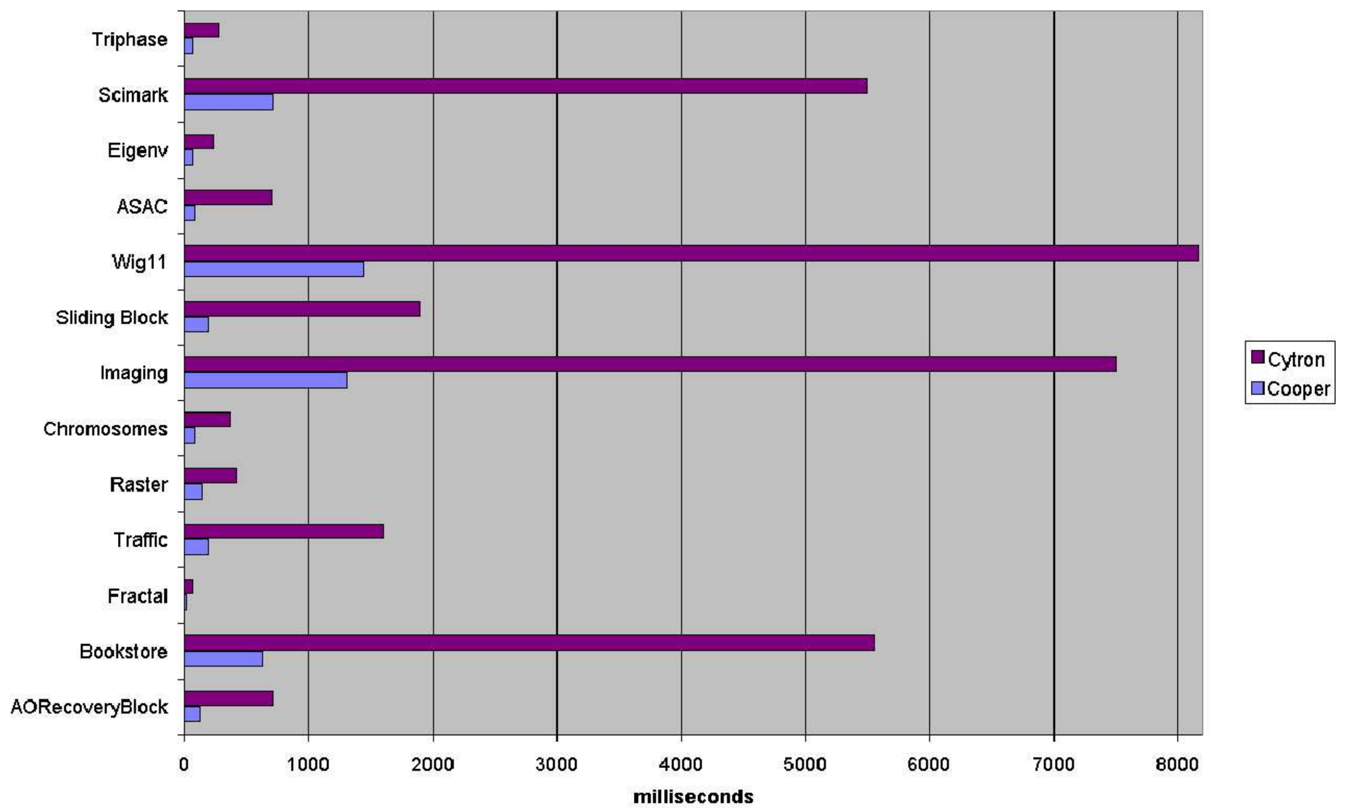

Figure 6: Cytron vs. Cooper Dominators, Dominator Tree, and Dominance Frontiers Algorithms.

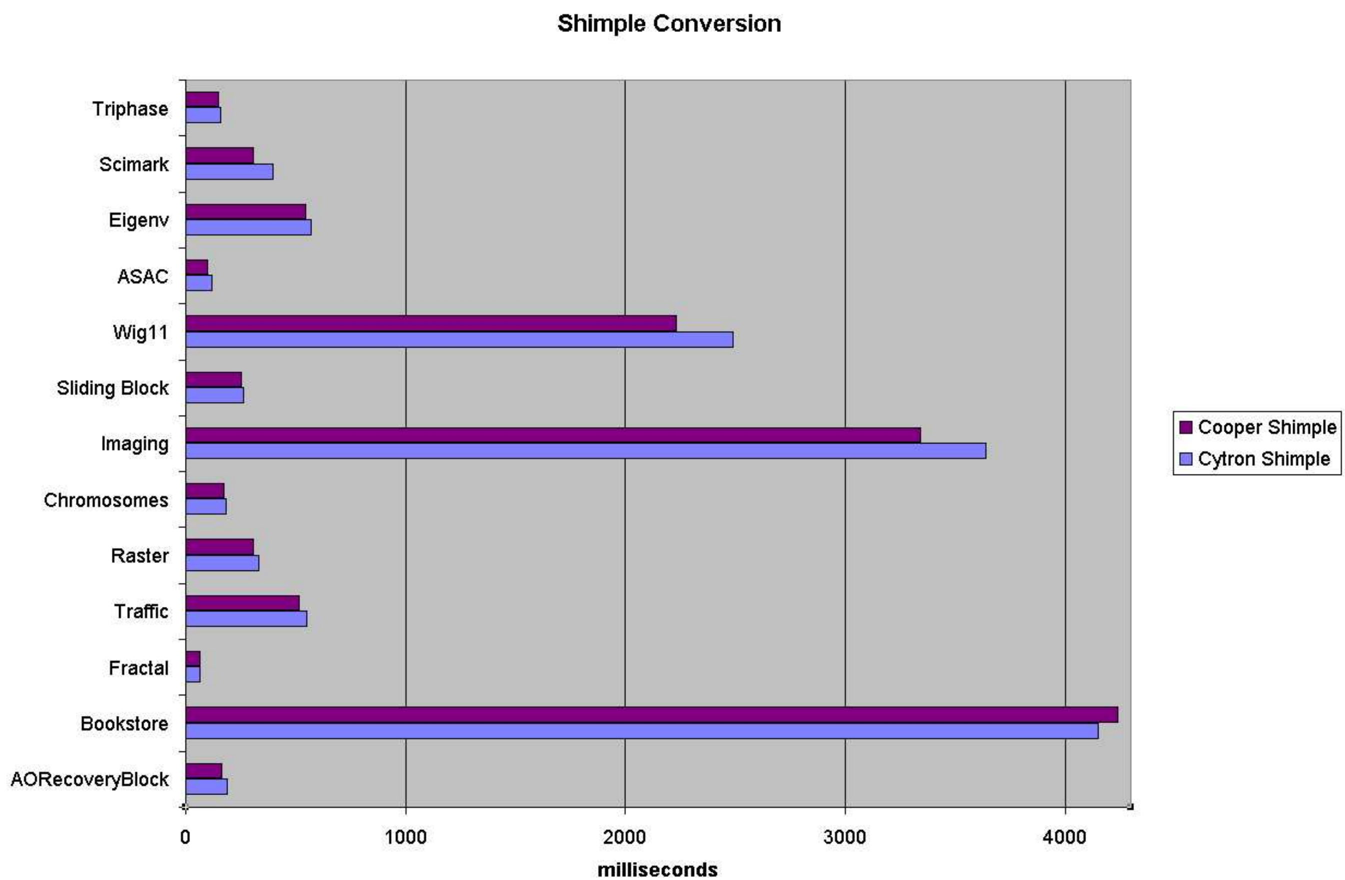

Figure 7: Jimple to Shimple conversion using Cytron vs. Cooper.

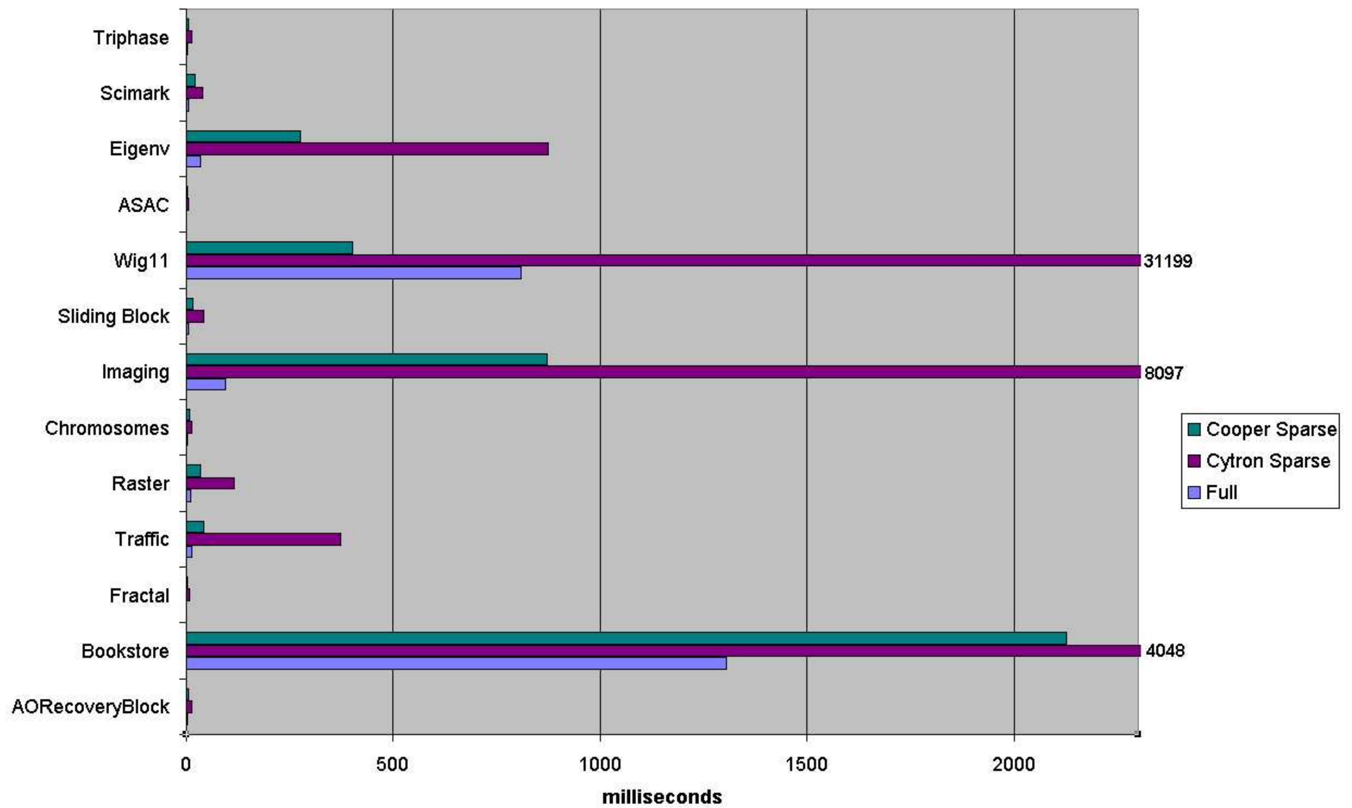

Figure 8: Forward Flow Analysis, Full vs. Cytron Sparse vs. Cooper Sparse.

The real meat of the results lies in the flow analysis comparisons. The lurking question is whether the sparse graphs created are small enough, and can be created fast enough, to result in a faster overall flow analysis. Unfortunately, as noted early, much of the benefit of sparse graphs is lost when there is no constant transference in your flow analysis. Results are disappointing for both forward and backward flow analyses, as seen in 8 and 9, respectively.

However, it is interesting to note that a few key benchmarks, wig11, bookstore, and traffic specifically, exhibit extreme blowups in both the full and sparse flow analysis. Some preliminary investigation into the cause of these blow ups points seems to point towards the use of very large switch statements.

The hand-optimized code was tested for speed improvements and the results were not very good. As shown in 10, there were speed improvements in all cases but none were able to break the double-digit percentage point improvement barrier. It would appear that javac/JVM combination is a fairly competent system that is able to perform much of the same optimizations (or at least produce similar running time to) as hand-optimizations by a programmer. The end of punch-card computing is far behind us, is hand-optimization and careful design choices heading the same way? Will optimizing compilers of the future be able to replace poor data structure choices for us in a seamless fashion? Clearly interesting questions.

After being exposed to so many poor looking graphs I began to wonder about the results. Some further testing yielded some *very* interesting results. Sparse flow analysis was not always resulting in fewer flow through and merge operations! Figure 11 shows, in fact, that it is somewhat arbitrary and inconsistent from one benchmark to another. The Wig11 compiler benchmark is truly interesting, though, with an oddly high number of merges compared to flow throughs - for both full and sparse analysis. This is most likely an example of the difficulty that flow analysis has with the switch construct, as my investigations led me to the

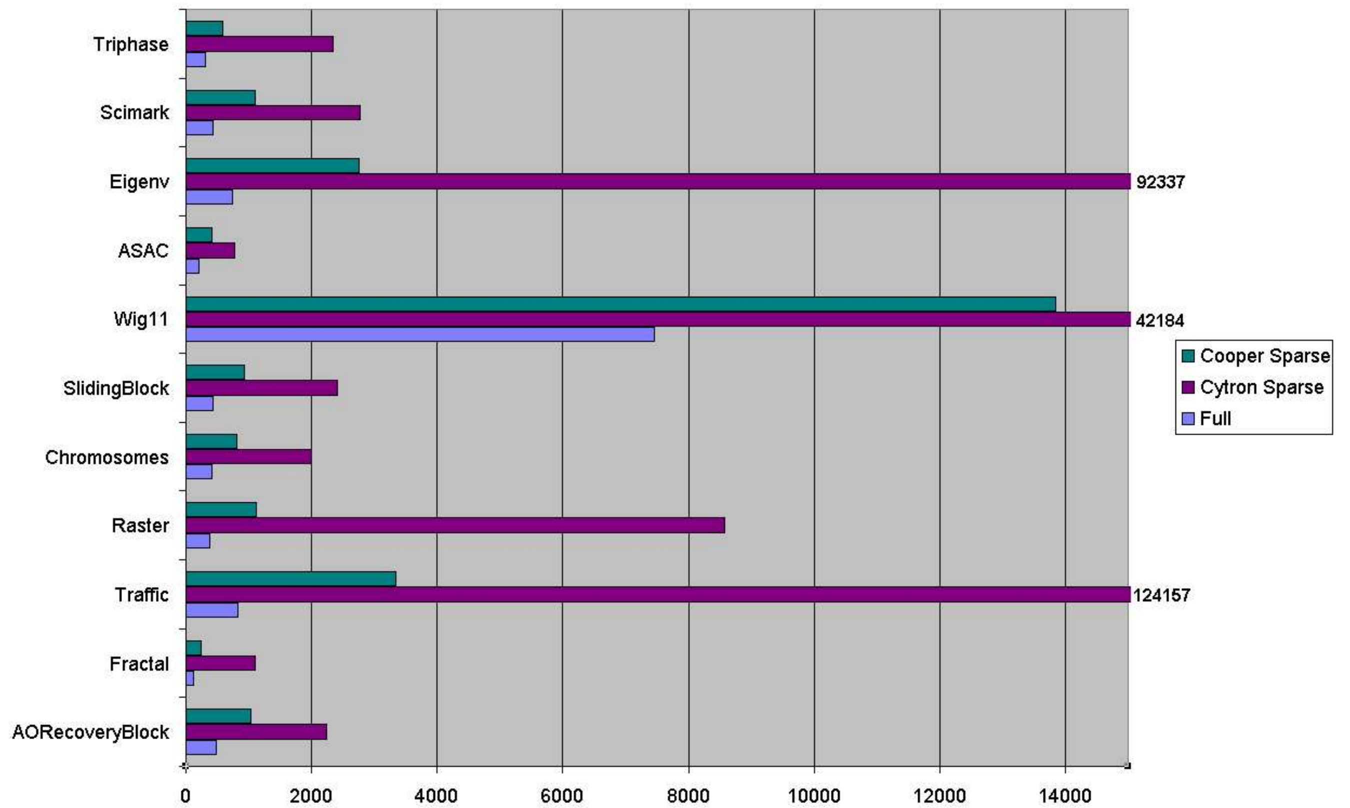

Figure 9: Backward Flow Analysis, Full vs. Cytron Sparse vs. Cooper Sparse

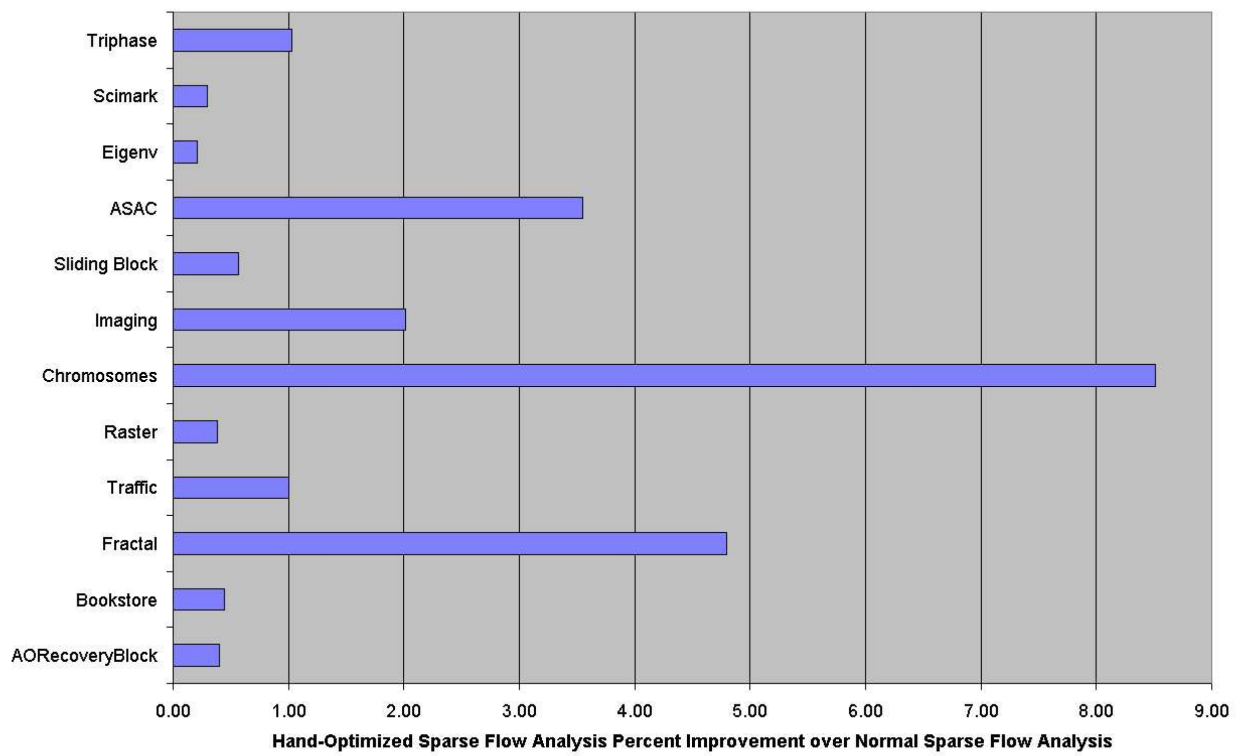

Figure 10: Improvement on Running Times after Hand Optimizations

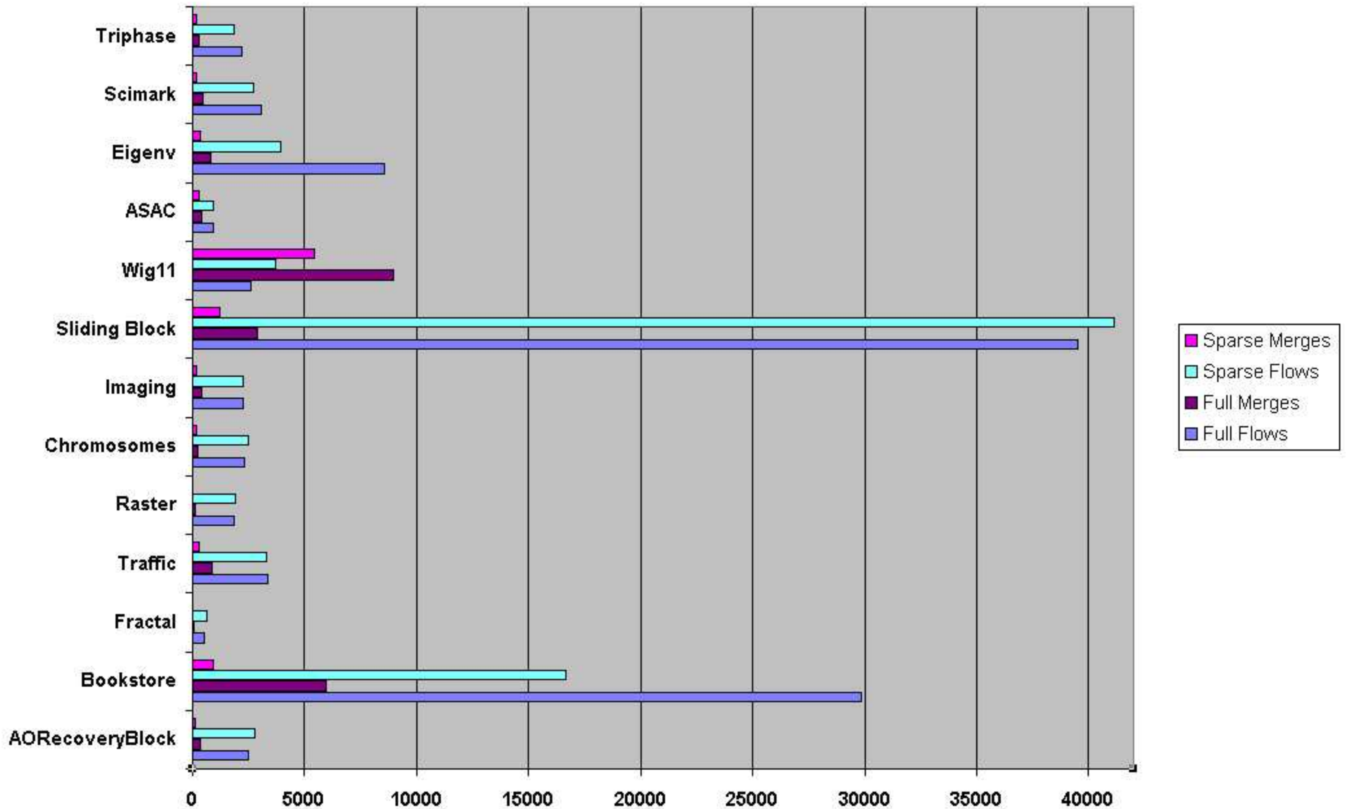

Figure 11: Number of Flow Through and Merge Operations performed, Full vs. Sparse Analysis

Lexer and *Parser* classes over and over when I tried to find the largest bottlenecks.

Finally, it would appear that merge operations are far more costly to a fixed-point iteration scheme than flow throughs. This can be seen by comparing the Sliding Block benchmark to the Wig11 benchmark. The Sliding Block benchmark has an astronomical blowup of flow throughs yet as can be seen in 8 and 9, it's flow analyses run fairly quickly. Wig11, however, has an inverse ratio of merges to flow throughs which results in terribly slow flow analysis.

The most likely reason that merges are so expensive in flow analyses is not that the operation of merging sets is so expensive but that the merges introduce the need for more iterations to reach a fixed point.

7 Conclusions and Future Work

I have designed and implemented a sparse data flow framework in Soot which closely models the existing code in Soot and is therefore easily integrated. I explored the use of this sparse data flow framework as a possible means to improve the data flow analysis performance. I successfully produced correct forward and backward flow analysis results from this sparse flow analysis framework, as tested against a suite of benchmark Java programs. Unfortunately, because the dominators, dominator tree, and dominance frontiers of a control flow graph must be computed in order to build a sparse flow graph, the approach did not yield faster results. Furthermore, the sparse graph implementation suffered due to my more general approach, which precludes any sense of constant transference - the idea that a node kills all information flowing into it during an analysis. The resulting sparse graphs built with this approach, while somewhat smaller than their full graph counterparts, are still far larger than what was initially hoped for. However, it appears that

the use of constant transference is a fairly limiting factor, only allowing for extremely specific flow analysis problems.

I implemented a simple and fast dominators algorithm which has a larger upper bound than some known methods yet proves to yield much better results overall when submitted to empirical testing. I also implemented an algorithm for computing the dominance frontier sets of a flow graph which is better, in practice, than other algorithms with smaller upper bounds. This further backs up the work of Cooper, et al [4], showing that sometimes algorithms with "discouraging asymptotic complexities" can be faster in practice than approaches that are engineered with smaller theoretical upper bounds.

Future work could include further investigation into why the sparse flow analyses that were run on the benchmark programs ultimately resulted in more iterations than their counterpart full flow analyses. Much thought was given to this peculiarity but it remains unclear why this is so.

In order to make use of sparse graphs more efficiently I introduced the notion of pre-computable dominators, dominator trees, and dominance frontiers which can be passed to each flow analysis instead of being recomputed. This same approach could be explored in the context of the sparse graph itself. For example, one could trim out all nodes but uses and definitions in order to build a conservative sparse graph valid for a wide set of flow analyses. Finally, the sparse flow analysis framework in Soot could be extended to handle branched flow analysis.

8 Acknowledgments

I would like to thank a fellow COMP-621 student, Kacper Wysocki, for his help during this project. He proved to be a wealth of information when it came to Unix shell scripting. He also provided an excellent sounding board for my ideas and had many useful suggestions of his own.

A Appendix

A.1 Source Code and Electronic Versions of this Paper

All source code mentioned in this paper is available at `http://www.cs.mcgill.ca/~mbatch/621/index.html` as a *tar.gz* file. Electronic copies of this paper are available there, as well, in postscript and pdf format.

A.2 FastDominatorTreeFrontier.class

This class contains the algorithms from Cooper, et al [4] which compute control flow graph dominators, dominator tree, and dominance frontiers. The alogrithms are completely contained in the class constructor and the intersect method. All other methods are simple accessor methods.

```
package soot.toolkits.graph;

import java.util.*;

/**
 * @author Michael Batchelder (mbatch@cs.mcgill.ca)
 */
public class FastDominatorTreeFrontier
{

  protected DirectedGraph graph;
  protected List heads;
  protected Map nodeToDominators;
```

```
protected Map nodeToIDominator;
protected Map nodeToFrontier;
protected Map nodeToDomChildren;
protected Map unitsToNumbers;

/*
 *  Fast Dominators Finder algorithm as explained in:
 *  "A Simple, Fast Dominance Algorithm" by Cooper, Harvey, and Kennedy
 *
 * NOTE: "doms[b] refers to the immediate dominator of b
 *
 *
 *  for all nodes, b // initialize the dominators array
 *          doms[b] = Undefined
 *  doms[start node] =  start node
 *  Changed = true
 *  while (Changed)
 *          Changed = false
 *          for all nodes, b, in reverse postorder (except start node)
 *                  new idom = first (processed) predecessor of b // (pick one)
 *                  for all other predecessors, p, of b
 *                          if doms[p] != Undefined
 *                          // i.e., if doms[p] already calculated
 *                                  new idom = intersect(p, new idom)
 *                  if doms[b] != new idom
 *                          doms[b] = new idom
 *                          Changed = true
 *
 *
 *  function intersect(b1, b2) returns node
 *          finger1 = b1
 *          finger2 = b2
 *          while (finger1 != finger2)
 *                  while (finger1 < finger2)
 *                          finger1 = doms[finger1]
 *                  while (finger2 < finger1)
 *                          finger2 = doms[finger2]
 *          return finger1
 *
 */
public FastDominatorTreeFrontier (DirectedGraph graph)
{
  this.graph = graph;
  int index = graph.size () * 2 + 1;
  ArrayList workList = new ArrayList (index);

  // precompute hash size
    index = index * 2 + 1;

  HashSet frontierList = new HashSet (index, 0.7f);
    nodeToIDominator = new HashMap (index, 0.7f);
    nodeToDominators = new HashMap (index, 0.7f);
    nodeToDomChildren = new HashMap (index, 0.7f);
```

```
  nodeToFrontier = new HashMap (index, 0.7f);
  unitsToNumbers = new HashMap (index, 0.7f);

  heads = graph.getHeads ();
Iterator uIt = heads.iterator ();
while (uIt.hasNext ())
  {
    Object u = uIt.next ();
    nodeToIDominator.put (u, u);
      workList.add (0, u);
    for (Iterator sIt = graph.getSuccsOf (u).iterator (); sIt.hasNext ();)
      if (!workList.contains ((u = sIt.next ())))
        workList.add (u);
  }

index = 0;
for (uIt = (new PseudoTopologicalOrderer ().newList (graph)).iterator ();
     uIt.hasNext ();)
  {
    Object u = uIt.next ();
    nodeToFrontier.put (u, new ArrayList ());

    // The Dominator children should not include SELF.
    nodeToDomChildren.put (u, new HashSet ());
    unitsToNumbers.put (u, new Integer (index++));
    if (graph.getPredsOf (u).size () > 1)
      frontierList.add (u);
  }

index = heads.size ();
while (workList.size () > index)
  {
    Object u = workList.get (index);
    Object iDom = null;
    List preds = graph.getPredsOf (u);
    if (preds.size () == 1)
      {
        if (nodeToIDominator.get (u) == null)
          {
            Object parent = preds.get (0);
            nodeToIDominator.put (u, parent);
            ((HashSet) nodeToDomChildren.get (parent)).add (u);
            uIt = graph.getSuccsOf (u).iterator ();
            while (uIt.hasNext ())
              {
                Object child = uIt.next ();
                if (!workList.contains (child))
                  workList.add (child);
              }
          }
      }
    else if (preds.size () > 1)
      {
```

```
        uIt = preds.iterator ();
        while (uIt.hasNext ())
          {
            Object pDom = uIt.next ();
            if (nodeToIDominator.get (pDom) != null)
              {
                if (iDom == null || iDom == pDom)
                  iDom = pDom;
                else
                  iDom = intersect (iDom, pDom);
              }
          }
        if (iDom != null)
          {
            Object oldDom = nodeToIDominator.get (u);
            if (oldDom != iDom)
              {
                nodeToIDominator.put (u, iDom);
                ((HashSet) nodeToDomChildren.get (iDom)).add (u);
                uIt = graph.getSuccsOf (u).iterator ();
                while (uIt.hasNext ())
                  {
                    Object child = uIt.next ();
                    if (!workList.contains (child))
                      workList.add (child);
                  }

                if (oldDom != null)
                  {
                    ((HashSet) nodeToDomChildren.get (oldDom)).remove (u);
                  }
              }
          }
      }
    workList.remove (index);
  }

/*
   Fast Dominators Frontier algorithm as explained in:
   "A Simple, Fast Dominance Algorithm" by Cooper, Harvey, and Kennedy

   for all nodes, b
   if the number of predecessors of b >= 2
   for all predecessors, p, of b
   runner = p
   while runner != doms[b]
   add b to runners dominance frontier set
   runner = doms[runner]
 */

for (uIt = frontierList.iterator (); uIt.hasNext ();)
  {
    Object u = uIt.next ();
```

```
      Object uDom = nodeToIDominator.get (u);
      Iterator pIt = graph.getPredsOf (u).iterator ();
      while (pIt.hasNext ())
        {
          Object runner = pIt.next ();
          Object tmp = null;
          while (runner != null && runner != uDom
                 && runner != u && tmp != runner)
            {
              List df = (List) nodeToFrontier.get (runner);
              if (!df.contains (u))
                df.add (u);
              tmp = runner;
              runner = nodeToIDominator.get (runner);
            }
        }
    }

  // cleanup for GC
  unitsToNumbers = null;
}

private Object intersect (Object d1, Object d2)
{
  Object temp;
  int i1 = ((Integer) unitsToNumbers.get (d1)).intValue ();
  int i2 = ((Integer) unitsToNumbers.get (d2)).intValue ();
  do
    {
      while (i1 > i2)
        {
          if ((temp = nodeToIDominator.get (d1)) == d1)
            return d2;
          d1 = temp;
          i1 = ((Integer) unitsToNumbers.get (d1)).intValue ();
        }
      while (i2 > i1)
        {
          if ((temp = nodeToIDominator.get (d2)) == d2)
            return d1;
          d2 = temp;
          i2 = ((Integer) unitsToNumbers.get (d2)).intValue ();
        }
    }
  while (i1 != i2);

  return d1;
}

public DirectedGraph getGraph ()
{
  return graph;
}
```

```
  /**
   * Returns a list of dominators for the given node in the graph.
   *
   * Don't bother building this in the constructor: only if someone
   * asks for it!
   **/
public List getDominators (Object node)
{
  ArrayList doms = (ArrayList) nodeToDominators.get (node);
  if (doms == null)
    {
      doms = new ArrayList ();
      Object idom = node;
      do
        {
          doms.add (idom);
          node = nodeToIDominator.get (idom);
        }
      while (idom != node && (idom = node) != null);
      nodeToDominators.put (node, doms);
    }
  return doms;
}

  /**
   * True if "node" is dominated by "dominator" in the graph.
   **/
public boolean isDominatedBy (Object node, Object dominator)
{
  return getDominators (node).contains (dominator);
}

  /**
   * True if "node" is dominated by "dominator" in the graph.
   **/
public boolean isDominatorOf (Object dominator, Object node)
{
  return getDominators (node).contains (dominator);
}

  /**
   * True if "node" is dominated by all nodes in "dominators" in the graph.
   **/
public boolean isDominatedByAll (Object node, Collection dominators)
{
  return getDominators (node).containsAll (dominators);
}

public List getDominanceFrontierOf (Object node)
{
  return (List) nodeToFrontier.get (node);
}
```

```
  public Object getImmediateDominator (Object node)
  {
    if (heads.contains (node))
      return null;
    Object ret = nodeToIDominator.get (node);

    return nodeToIDominator.get (node);
  }

  public Iterator getChildrenOf (Object node)
  {
    return ((HashSet) nodeToDomChildren.get (node)).iterator ();
  }
}
```

A.3 FastSparseGraph.class

This class contains the algorithm from Choi, et al [3] for building a sparse data flow graph given a flow analysis domain. The code is optimized.

```
package soot.toolkits.graph;

import soot.*;
import soot.toolkits.scalar.*;
import java.util.*;

/**
 * @author Michael Batchelder (mbatch@cs.mcgill.ca)
 */
public class FastSparseGraph implements DirectedGraph
{

  protected List heads, tails;
  protected FastSparseFlowAnalysis flowAnalysis;
  protected boolean backwardFlow;
  protected DirectedGraph dg;
  protected FastDominatorTreeFrontier dtf;
  protected Map unitToSuccs;
  protected Map unitToPreds;
  protected Map fullToSparseUnit;
  protected HashSet nodes;

  //  Helper fields. Only for creation, then cleared out
  protected Stack buildStack;
  protected HashSet MeetNodes;
  private List TOSsuccs;
  private Object TOS;

        /**
         * Constructs a FastSparseGraph given a Body instance as outlined in:
         * "Automatic Construction of Sparse Data Flow Evaluation Graphs"
         * by Jong-Deook Choi, Ron Cytron, and Jeanne Ferrante.
```

```
      */
public FastSparseGraph (DirectedGraph g, FastSparseFlowAnalysis fA)
{
  dg = g;
  flowAnalysis = fA;
  backwardFlow = fA.isBackward ();
  heads = g.getHeads ();
  tails = g.getTails ();
  int size = g.size () * 2 / 3 + 1;

    nodes = new HashSet (size, 0.7f);
    MeetNodes = new HashSet (size, 0.7f);
    unitToSuccs = new HashMap (size, 0.7f);
    unitToPreds = new HashMap (size, 0.7f);
    fullToSparseUnit = new HashMap (size, 0.7f);
    dtf = new FastDominatorTreeFrontier (g);
  Iterator uIt = g.iterator ();
  while (uIt.hasNext ())
    {
      Unit u = (Unit) uIt.next ();

      // if already visited, skip
      if (nodes.contains (u))
          continue;
      else if (!flowAnalysis.isFlowThroughIdentity (u))
        {
          nodes.add (u);
          Iterator dfIt = dtf.getDominanceFrontierOf (u).iterator ();
          while (dfIt.hasNext ())
            {
              Object f = dfIt.next ();
              if (!MeetNodes.contains (f))
                {
                  MeetNodes.add (f);
                  nodes.add (f);
                }
            }
        }
    }

  // Even if heads/tails should not be in the sparse graph, add them
  // to it, because we always need an ultimate parent for every node
  // to retrieve it's in flow if it's not in the sparse graph itself.
  nodes.addAll (heads);
  nodes.addAll (g.getTails ());

  for (uIt = nodes.iterator (); uIt.hasNext ();)
    {
      Object u = uIt.next ();
      unitToPreds.put (u, new ArrayList ());
      unitToSuccs.put (u, new ArrayList ());
    }
}
```

```
public void initialize ()
{
  buildStack = new Stack ();

  // search all heads to build graph
  Search (heads.iterator ());

  // clear out for GC
  MeetNodes = null;
  buildStack = null;
  TOSsuccs = null;
  TOS = null;
  dg = null;
  dtf = null;
}

private void Search (Iterator uIt)
{
  while (uIt.hasNext ())
    {
      Object unit = uIt.next ();
      boolean inNodes = false;
      if (!MeetNodes.contains (unit))
        {
          if (heads.contains (unit))
            inNodes = true;
          else if (nodes.contains (unit))
            {
              // Inlined Link() code from below
              if (!TOSsuccs.contains (unit))
                TOSsuccs.add (unit);

              List preds = (List) unitToPreds.get (unit);
              if (!preds.contains (TOS))
                preds.add (TOS);
            }
        }
      else
        {
          inNodes = nodes.contains (unit);
        }

      // inNodes now indicates whether the unit is in the nodes list
      if (inNodes)
        {
          buildStack.push (unit);
          TOS = unit;
          TOSsuccs = (List) unitToSuccs.get (TOS);
        }
      else
        {
          fullToSparseUnit.put (unit, TOS);
```

```
        }

      Iterator sIt = dg.getSuccsOf (unit).iterator ();
      while (sIt.hasNext ())
        {
          Object u = sIt.next ();
          if (MeetNodes.contains (u) && !heads.contains (unit))
            {
              // BEGIN This is inlined code from the Link() method below

              /*
               * There is probably no point in including the
               * getFlowThroughConstant part of the sparse graph analysis
               * since we never know constant transference because we are
               * using the sparse graph in a general sense
               *
               * Object flowset =
               * flowAnalysis.getFlowThroughConstant((Unit)u);
               * if (flowset==null) {
               *
               */
              if (!TOSsuccs.contains (u))
                TOSsuccs.add (u);
              List preds = (List) unitToPreds.get (u);
              if (!preds.contains (TOS))
                preds.add (TOS);
              //}
              // END This is inlined code from the Link() method below
            }
          else
            {
              fullToSparseUnit.put (u, TOS);
            }
        }

      Search (dtf.getChildrenOf (unit));

      if (inNodes && buildStack.size () > 1)
        {
          buildStack.pop ();
          TOS = buildStack.peek ();
          TOSsuccs = (List) unitToSuccs.get (TOS);
        }
    }
}

/*
 * This Method Has Been INLINED into SEARCH. SORRY!
 *
 * private void Link(Object unit) {
 *  Object flowset = flowAnalysis.getFlowThroughConstant((Unit)u);
 *  if (flowset==null) {
 *    // If there is NOT constant transference
```

```
 *    // then we need to add it to the succs/preds
 *    if (!TOSsuccs.contains(unit)) TOSsuccs.add(unit);
 *    List preds = (List)unitToPreds.get(unit); if (!preds.contains(TOS))
 *    preds.add(TOS);
 *  }
 * }
 */

public Object fullUnitToSparseUnit (Object unit)
{
  return fullToSparseUnit.get (unit);
}

public List getHeads ()
{
  return heads;
}

public List getTails ()
{
  return tails;
}

public List getPredsOf (Object s)
{
  return (List) unitToPreds.get (s);
}

public List getSuccsOf (Object s)
{
  return (List) unitToSuccs.get (s);
}

public int size ()
{
  return nodes.size ();
}

public Iterator iterator ()
{
  return nodes.iterator ();
}

public String toString ()
{
  Iterator it = nodes.iterator ();
  StringBuffer buf = new StringBuffer ();
  while (it.hasNext ())
    {
      Unit u = (Unit) it.next ();
      buf.append ("// preds: " + getPredsOf (u) + "\n");
      buf.append (u.toString () + '\n');
      buf.append ("// succs " + getSuccsOf (u) + "\n");
```

```
    }
  return buf.toString ();
 }

 public boolean contains (Object unit)
 {
   return nodes.contains (unit);
 }
}
```

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[2] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.

[3] Jong-Deook Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '98: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66. ACM Press, 1991.

[4] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[7] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[8] Reese Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138, NY, USA, December 1959. Spartan Books.

[9] McGill University Sable Research Group. Soot source code and Soot API. http://www.sable.mcgill.ca/soot.

[10] Jack Shirazi. *Java performance tuning*. Java series. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2000.